Processes
Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data segment
  - instructions
Process in Memory
Process State

- As a process executes, it changes state
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution
Diagram of Process State

- New
- Admitted
- Ready
- Running
- Waiting
- Terminated
- Exit
- I/O or event completion
- Scheduler dispatch
- I/O or event wait

Processes
Process Control Block (PCB)

Information maintained by the kernel (metadata) associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information
Process Control Block (PCB)

- process state
- process number
- program counter
- registers
- memory limits
- list of open files
  

...
CPU Switch From Process to Process

- **Process $P_0$**
  - Executing
  - Interrupt or system call
  - Save state into PCB$_0$
  - ... ($\ldots$)
  - Reload state from PCB$_1$

- **Operating System**
  - Idle
  - ... ($\ldots$)

- **Process $P_1$**
  - Executing
  - Interrupt or system call
  - Save state into PCB$_1$
  - ... ($\ldots$)
  - Reload state from PCB$_0$

- **Idle**
Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues
Ready Queue And Various I/O Device Queues
Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into memory, and thus, into the ready queue
- **Medium-term scheduler** – moves partially executed processes to/from disk storage to adjust the degree of multiprogramming
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
Addition of Medium Term Scheduling

Medium term scheduler

- swap in
- partially executed swapped-out processes
- swap out

- ready queue
- CPU
- I/O
- I/O waiting queues

Long term Scheduler
Schedulers (Cont.)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Medium-term scheduler is invoked less frequently
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- The medium-term and long-term schedulers control the degree of multiprogramming
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
Context Switch

• When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process

• Context-switch time is overhead; the system does no useful work while switching

• Time dependent on hardware support
Process Creation

- Parent processes create child processes, which, in turn create other processes, forming a tree of processes
- Possible kinds of resource sharing
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it

- UNIX/Linux examples
  - `fork` system call creates new process
  - `exec` system call used after a `fork` to replace the process’ memory space with a new program
Process Creation

![Diagram of process creation]

- **fork()**: Creates a new process (child). The parent process continues to run.
- **exec()**: Replaces the current program with a new program. The child process continues to run.
- **exit()**: Terminates the process.
- **wait()**: Waits for a child process to complete.
- **resumes**: Continues execution after a waiting process completes.

The diagram shows the flow of control between these functions, indicating the lifecycle of a process from creation to termination.
C Program Forking Separate Process

```c
int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execvp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait (NULL);
        printf ("Child Complete");
        exit(0);
    }
}
```
Process Termination

- Process executes last statement and asks the operating system to delete it (exit)
  - Output data from child to parent (via wait)
  - Process’ resources are deallocated by operating system

- Parent may terminate execution of children processes (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - cascading termination
Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process.
- **Cooperating** process can affect or be affected by the execution of another process.
- Advantages of process cooperation:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
Producer-Consumer Problem

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
  - unbounded-buffer places no practical limit on the size of the buffer
  - bounded-buffer assumes that there is a fixed buffer size
Bounded-Buffer – Shared-Memory Solution

- Shared data

  ```
  #define BUFFER_SIZE 10
  typedef struct {
      ...
  } item;
  
  item buffer[BUFFER_SIZE];
  int in = 0;
  int out = 0;
  ```

- The following solution is correct, but can only use BUFFER_SIZE-1 elements
Bounded-Buffer – Insert() Method

```c
while (true) {
    /* Produce an item */
    while (((in + 1) % BUFFER_SIZE) == out) {
        /* do nothing -- no free buffers */
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
    }
}
```
while (true) {
    while (in == out) {
        /* do nothing -- nothing to consume */
    }

    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    return item;
}